A Language for the Composition of Privacy-Enforcement Techniques

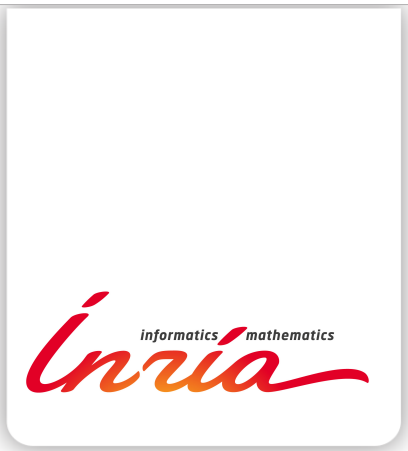
Ronan-Alexandre Cherrueau, Rémi Douence, Mario Südholt

A Language for the Composition of Privacy-Enforcement Techniques

Ronan-Alexandre Cherrueau, Rémi Douence, Mario Südholt

RESEARCH REPORT
N° 8720
April 2015
Project-Team ASCOLA

ISSN 0249-6399 ISRN INRIA/RR--8720--FR+ENG



A Language for the Composition of Privacy-Enforcement Techniques

Ronan-Alexandre Cherrueau *, Rémi Douence*, Mario Südholt*

Project-Team ASCOLA

Research Report n° 8720 — April 2015 — 17 pages

Abstract:
Today's large-scale computations, *e.g.*, in the Cloud, are subject to a multitude of risks concerning the divulging and ownership of private data. Privacy risks are mainly addressed using a large variety of encryption-based techniques. However, these are costly to operate, lead to large aggregates of data that are highly valuable attack targets and do not allow to flexibly handle subsets of such aggregates. Furthermore, today's computations have to ensure privacy properties in the context over highly variable and complex software compositions; however, no general support for the declarative definition and implementation of privacy-preserving applications has been put forward.
This article presents a compositional approach to the declarative and correct composition of privacy-preserving applications in the Cloud. Our approach provides language support for the compositional definition of encryption- and fragmentation-based privacy-preserving algorithms. This language comes equipped with a set of laws that allows us to verify privacy properties. Finally, we introduce implementation support in Scala that ensures certain privacy properties by construction using advanced features of Scala's type system.

Key-words: Language, Fragmentation, Encryption, Client-side computation, Typing, Algebraic laws

* École des Mines de Nantes, Nantes, France: `firstname.lastname@mines-nantes.fr`

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Langage pour la composition des techniques de protection de la vie privée

Résumé :

Aujourd'hui, et notamment à cause de l'Informatique en Nuage, les calculs à larges échelles sont risqués pour le respect de la vie privée. Une manière généralement utilisée pour garantir la vie privée est d'utiliser les techniques de chiffrement qui rendent les données privées lisibles que par les personnes autorisées. Elles représentent une protection très efficace lorsque seul le stockage est considéré. Mais, leur emploi devient très couteux lorsque d'autres formes de calculs doivent être effectuées sur les données chiffrées. L'utilité des techniques de chiffrement est donc limitée pour les systèmes à larges échelles.

Nous considérons que le protection de la vie privée dans les systèmes à larges échelles est possible. Mais, les développeurs d'applications ne doivent pas se limiter aux techniques de chiffrement. Ils doivent envisager la composition des différentes techniques de protection de la vie privée. Cet article présente un langage pour la composition correcte des techniques de protection de la vie privée. Le langage est équipé de lois algébriques qui nous permettent de vérifier le respect de la vie privée. Enfin, nous proposons un support de programmation en Scala pour notre langage. Le support utilise le système de type de Scala pour assurer la composition correcte.

Mots-clés : Langage, Fragmentation, Chiffrement, Calculs côté client, Typage, Lois algébriques

1 Introduction

The generalization of large-scale service-based computations executed over mutualized resources, notably in the context of Cloud computing, has considerably increased the risk of losing control or even ownership of one's personal data. In particular, the confidentiality and integrity of private data are at risk. Currently, privacy-preserving computations use encryption techniques in order to preserve such properties of private data. However, these techniques have important drawbacks. They are costly to apply. They result in much data being kept in one place (being a prime target for attacks). And they do not allow to flexibly handle subsets of encrypted data.

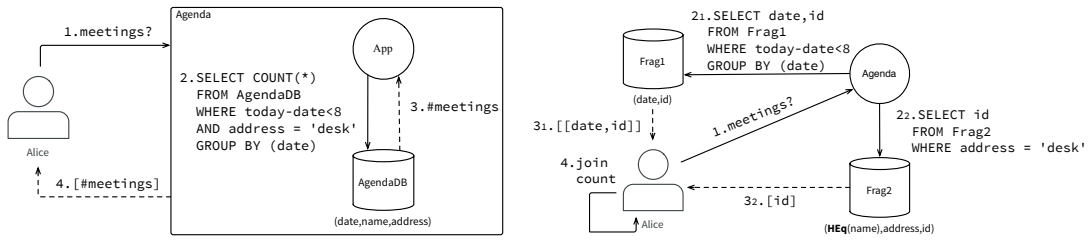
In order to improve on these characteristics, alternative approaches have been explored. Data fragmentation [ABG+05, dVEF+13], in particular, consists in dividing data sets into different parts and store them in different places. No unauthorized party can thus extract sensitive information from the pieces. Fragmentation allows to eliminate (much of) the computational overhead incurred by en/decryption. It allows the distribution of data on a multitude of sites and supports the handling of subsets of data sets. Because fragmentation-based approaches frequently use encryption for parts of the data and computations, handling privacy properties results in complex compositions of privacy techniques. However, no comprehensive composition approach for the construction of privacy-preserving computations has been put forward until today.

Figures 1a and 1b show the need for such a comprehensive composition approach for the construction of privacy-preserving computations. The first illustrates a privacy-preserving query (of the number, per day, of meetings Alice had in her office last week) on a local application. The query is easy to formulate because Alice's data is stored locally and is not subject to privacy problems. However, the same computation in the context of cloud computing requires to encrypt the database. Then, Alice has to decrypt the database at her side to perform the query and encrypt the result once again, which is obviously not efficient. The second illustration shows how the composition of encryption with fragmentation can improve the query efficiency. The database is fragmented so that no privacy concerns can be violated: many queries can thus operate without any decryption overhead. However, in this case the formulation of queries is far more difficult and error-prone.

Currently, compositions of privacy-ensuring strategies are programmed using traditional programming means (languages or frameworks) that do not provide any correctness guarantees. This state-of-affairs is, however, clearly unsatisfying, leads to numerous privacy violations in the real world, and incurs steep costs for individuals as for the society as a whole.

In this paper, we introduce the following contributions:

- A motivation for the need of dedicated means for the correct composition of different privacy-enforcing strategies, in particular, strategies based on encryption, fragmentation and client



(a) Local Application (b) Cloud Application

Figure 1: Privacy-Preserving Agenda as Local (a) and Cloud (b) Application

side computation. (Sec. 2)
- A language for the declarative definition of a range of (composed) strategies for the enforcement of privacy-centric properties based on encryption and fragmentation algorithms. The language comes equipped with a set of laws that ensure a number of properties ensuring privacy-properties. (Sec. 3)
- An implementation of the language in Scala that harnesses advanced typing properties in order to support the enforcement of privacy properties. (Sec. 4)

In addition, the paper discusses related work (Sec. 5) and provides a conclusion and future works (Sec. 6).

2 Motivation

Consider an application developed for the Cloud in order to improve, for instance, its availability, data replication properties or integration with other applications. Typically, the basic workflow consists in first the outsourcing of user data to a database service, and then applying applications hosted in the cloud to that data. For the case of the agenda application, one has first to subscribe to a cloud database service and outsource meeting data, and then install the agenda application on a Cloud platform.

The privacy concern In the age of cloud computing, cloud application programmers outsource without any consideration personal data of individuals. Outsourcing reveals personal data to cloud providers that can share information with third parties. This represents the Achilles' heel of cloud computing: how to share data and keep personal data private.

To handle the privacy concern, two types of approaches have been introduced. The first focuses on supervised cloud. It includes approaches relying on access control and policy enforcement [PM11, CJL12] to support accountability for violations [WABL+08]. The second focuses on the unsupervised cloud. It is a set of techniques to keep data private, such as encryption [Sal03], fragmentation [dVEF+13], differential privacy [RP10] and client side computation [FKDL13]. Each technique has pros and cons but, in general, they are limited, and do not offer as many guarantees as policy enforcement. For instance, data encryption fits for storage protection but not for computations and differential privacy is useful for statistical databases only. The main advantage is that they are applicable to real world problems.

This article handles the privacy concern in unsupervised clouds. We show that even if each technique taking by itself is limited, the composition of such techniques is much more effective and enables the development of expressive privacy-preserving cloud applications. In the remainder of this article, we will focus on the composition of three classes of techniques: encryption, fragmentation and client-side computations.

2.1 Encryption, Fragmentation and Client-Side Computations

Encryption Encryption [Sal03] is the process of encoding information before it is outsourced in such a way that only authorized parties can read it.

Historically, encryption is the first approach to the protection of private databases in the Cloud. Using, for instance, a symmetric encryption algorithm, a client encrypts its data before outsourcing it in the cloud. All queries can then simply be executed by first returning the necessary data from the database to the client in its encrypted state, be decrypted by her, and execute the query at her side. This approach is, however, far too expensive to be practical. One solution to this problem are recent encryption schemes, so called homomorphic ones, that execute query directly on encrypted data.

Theoretically, fully homomorphic schemes enable arbitrary operations to be performed on encrypted data. However, these schemes are prohibitively expensive. Reasonably efficient homomorphic schemes are currently known only for a small set of operations. A deterministic encryption scheme, *e.g.*, permits efficiently to check the equality of values by comparing encrypted data. For this reason, query execution over encrypted data is often seen as practical only if corresponding efficient homomorphic encryption schemes are available [NLV11].

Fragmentation (Vertical) fragmentation [ABG+05, dVEF+13] is the process of separating information into non-linkable fragments in such a way that only authorized parties can recompose the original information. Fragmentation is applicable when associations of data are sensitive rather than individual data items themselves.

Fragmentation is tightly coupled to the notion of privacy constraints. A privacy constraint specifies which data is sensitive and should, therefore, be kept confidential. In our agenda application, for instance, meetings should satisfy two privacy constraints. An agenda meeting is the triplet $(date, name, address)$ that represents the meeting date, the name of the contact and the meeting location. The first constraint is $\{date, address\}$ so that an attacker cannot localize Alice by associating an address to a meeting date. The second constraint should be $\{name\}$ so that an attacker cannot infer the name of Alice's contacts. Here, fragmentation aims to make privacy associations such as $\{date, address\}$ safe by splitting the triplet $(date, name, address)$ in two.

Client-side computation Client-side computation [FKDL13] designates the concept of letting clients perform sensitive computations on their own and upload only the results. Especially, client side computation stores private information at the client side. Thus, computations on private data are performed at the client side. Services that require strong guarantees on the truthfulness of the result, such as billing service, can use, *e.g.*, zero-knowledge protocols to ensure the integrity of the query results and privacy.

2.2 Composition of Privacy-Enforcement Techniques

The application Figure 1b shows a common case in which a composition of techniques is required to obtain efficient and private application in the cloud. The application is the cloud version of the local the one (Fig. 1a) and requires a composition of encryption, fragmentation and client-side computation.

To distribute this agenda application in a privacy-preserving manner, the programmer adopts the two privacy constraints $\{date, address\}$ and $\{name\}$ introduced in Sec. 2.1. A programmer then has to choose a configuration for the application that satisfies the privacy constraints. First, she may, *e.g.*, fragment the database in two. The first fragment contains the dates. The second contains the names and addresses. Now, the $\{date, address\}$ constraint is safe unless the two fragments are joined. Then, the programmer encrypts names with an homomorphic encryption that supports equality, thus ensuring that the $\{name\}$ constraint is satisfied.

Based on such a configuration, *e.g.*, the query that computes the number of meetings Alice had per day at her desk last week (1 in Fig. 1b) is distributed on both fragments. It applies selection and grouping operations on the first fragment (2_1) to obtain the dates and identifiers of the meetings of the past week. At the same time, it applies the selection on the second fragment (2_2) to get identifiers of the office meetings. The application then fully harnesses the cloud, whether for storage or querying. However, the agenda has to join results of both fragments to count the number of people. Because that operation is not privacy preserving, finally, the rest of the query is executed at Alice's side (3),(4).

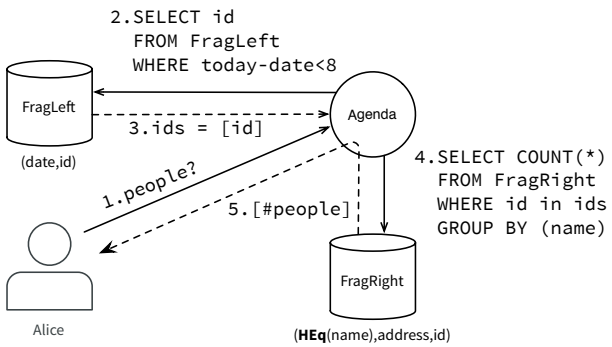


Figure 2: Number of people Alice met last week

Another useful query built on top of cloud configuration is the number of people Alice has met last week (Fig. 2). The query uses a *left-first* strategy of fragmentation to be efficient. It applies the selection on the left fragment to get identifiers of meetings the past week (2), (3). It then applies projection and grouping operations to the right fragment. And the programmer uses identifiers obtained from the left fragment to reduce the number components (4). The grouping operation requires to compare names, but this is fine since names are encrypted with an homomorphic scheme that supports equality testing. Moreover, the overhead due to encryption is minimal because the selection largely reduces the number of comparisons.

2.3 Language-Support to the Rescue

The above examples, as well as the large majority of real Cloud applications, require several privacy techniques to be composed. In this case, errors easily slip in. For instance, trying to encrypt twice the same column in a database does not make sense. Similarly, trying to sum values encrypted with a symmetric scheme is not reasonable.

To help programmers composing privacy techniques, we propose a functional language that focuses on privacy protection and query computation. As input, we have a privacy relation and apply privacy/query functions in turn to reduce the number of components in the relation. Together with the language, a set of algebraic laws specifies how to transform a local program to a privacy-preserving distributed Cloud application.

We are convinced that this formalization constitutes an appropriate abstraction to help programmers reason on query and privacy techniques composition. The corresponding language-level and implementation support the declarative definition of privacy-preserving Cloud applications and their efficient execution.

3 Language-based composition of privacy-enforcement techniques

This section defines our language for composition of encryption, fragmentation and client side computation techniques. First, we introduce its commands based on a SQL-like query language extended with abstractions for fragmentation and encryption. Second, we show how a query can be transformed by introducing privacy commands. Finally, we formalize and generalize our approach by corresponding composition laws.

3.1 Language Description

Our language is based on database queries obeying relational algebra properties.

A relation (*i.e.*, a table) is a set of tuples. For instance, an agenda stores a meeting as a triple $(date, name, address)$ representing the meeting date, the name of the contact and the meeting location. Our language offers four query functions:

- A selection σ filters tuples of a relation. For instance, $lastWeekAtDesk = \sigma_{(today-date<8)\wedge(address=desk)}$ keeps meetings whose date is at most a week old and has desk as meeting venue.
- A projection π keeps a subset of the columns of a relation. For instance $days = \pi_{date}$ keeps only the date of meetings.
- A grouping definition *group* groups together tuples of a relation. For instance, $byDay = group_{date}$ creates a group of tuples for each date.
- An aggregation $fold$ computes a single value for groups. For instance, $count = fold\ (+1)\ 0$ counts the number of meeting in a group.

Note that we omit a product operation of tables because it is not required by our examples but it could be easily introduced. These functions can be composed ($\circ$) in order to define complex queries according to the following grammar:

$$Q ::= Q \circ Q \mid \sigma \mid \pi \mid group \mid fold\ f\ k$$

Note that a sequence of function compositions is read from right to left. For instance, the number of meetings per day at desk last week is:

$$\#meeting = count \circ byDay \circ days \circ lastWeekAtDesk$$

Our language also provides privacy-related pairs of functions:

- $crypt_{\mathsf{s},as}/decrypt_{\mathsf{s},as}$ encrypts/decrypts the components of tuples corresponding to π_{as}. For instance, $cryptContact = crypt_{\mathsf{heq},name}$ encrypts the contacts in the agenda, although they still can be compared because heq is an homomorphic encryption that supports equality.
- $frag_{\pi_{as}}/defrag_{\pi_{as}}$ vertically fragments/defragments a table of tuples into two tables of tuples, so that the tuples of the first table contain only the components corresponding to π_{as} and the tuples of the second table contain the remaining components (noted $\pi_{\bar{as}}$). For instance, $fragDate = frag_{\pi_{date}}$ fragments the agenda table into a first table for the dates and a second table for the names and addresses. Remember [ABG+05] that in both tables each tuple also contains an index in order to reconstruct the original tuples.
- $frag_{\sigma_p}/defrag_{\sigma_p}$ horizontally fragments/defragments a table of tuples into two tables. For instance, $frag_{lastWeek}$ fragments the agenda into two tables of meetings. The first table contains only the meetings of last week and the second table contains the older meetings.

These privacy functions can be composed to make queries privacy-aware as defined by the following grammar:

$$Q_p ::= Q_p \circ Q_p \mid Q \mid crypt \mid decrypt \mid frag \mid defrag$$

For instance, when the agenda is fragmented (*date* on one host, *name* and *address* on another host) and the identity of contacts is encrypted, the query for the number of meetings per day at desk last week is:

$$\begin{aligned} \#meetingPrivate = \; & count \circ defragDate \\ & \circ (byDay \circ days \circ lastWeek, atDesk) \\ & \circ fragDate \circ cryptContact \end{aligned}$$

It can be read as: encrypt the contacts' names, then fragment the table, select the meeting dates of last week on the first fragment and group them, select the meeting with desk as venue on the second fragment, defragment the results in order to get a group by day but only for desk venue, and finally count the number of groups (*i.e.*, meetings per day). Note that there is no need to call the decrypt function here because *count* does not require values, but *count* must be executed at client side because $defrag$ discards protection.

The next section shows how to transform the centralized query $\#meeting$ into the private one $\#meetingPrivate$.

3.2 Making Private a Query with Transformations

The centralized query that computes the number of meeting per day at desk last week is:

$$Q_1 \equiv count \circ group_{date} \circ \pi_{date} \circ \sigma_{(today-date<8)\wedge(address=desk)}$$

It can be decentralized by introducing privacy-related functions. First, we add $frag$ then $defrag$ at the beginning (*i.e.*, right) of the query. This is correct since these functions are inverse

$$\begin{aligned} \equiv \; & count \circ group_{date} \circ \pi_{date} \circ \sigma_{(today-date<8)\wedge(address=desk)} \\ & \circ \; defrag_{\pi_{date}} \circ frag_{\pi_{date}} \end{aligned}$$

Similarly we introduce encryption

$$\begin{aligned} \equiv \; & count \circ group_{date} \circ \pi_{date} \circ \sigma_{(today-date<8)\wedge(address=desk)} \\ & \circ \; defrag_{\pi_{date}} \circ frag_{\pi_{date}} \\ & \circ \; decrypt_{\mathsf{heq},name} \circ crypt_{\mathsf{heq},name} \end{aligned}$$

decrypt and $defrag$ must be executed at the owner's place since they discard protection, so we delay them to the left. $defrag$ commutes with σ, but the selection must be applied only to the relevant fragment

$$\begin{aligned} \equiv \; & count \circ group_{date} \circ \pi_{date} \circ defrag_{\pi_{date}} \\ & \circ \; (\sigma_{(today-date<8)}, \sigma_{(address=desk)}) \\ & \circ \; frag_{\pi_{date}} \circ decrypt_{\mathsf{heq},name} \circ crypt_{\mathsf{heq},name} \end{aligned}$$

$defrag$ and π commutes, but the selection must be applied to each frament

$$\begin{aligned} \equiv \; & count \circ group_{date} \circ defrag_{\pi_{date}} \\ & \circ \; (\pi_{date} \circ \sigma_{(today-date<8)}, \pi_{date} \circ \sigma_{(address=desk)}) \\ & \circ \; frag_{\pi_{date}} \circ decrypt_{\mathsf{heq},name} \circ crypt_{\mathsf{heq},name} \end{aligned}$$

The projection on components missing in the fragment can be simplified

$$\begin{aligned}
&\equiv count \circ group_{date} \circ defrag_{\pi_{date}} \\
&\quad \circ \ (\pi_{date} \circ \sigma_{(today-date<8)}, \sigma_{(address=desk)}) \\
&\quad \circ \ frag_{\pi_{date}} \circ decrypt_{\mathsf{heq},name} \circ crypt_{\mathsf{heq},name}
\end{aligned}$$

defrag and *group* commutes, but grouping must be performed only in the relevant fragment

$$\begin{aligned}
&\equiv count \circ defrag_{\pi_{date}} \\
&\quad \circ \ (group_{date} \circ \pi_{date} \circ \sigma_{(today-date<8)}, \sigma_{(address=desk)}) \\
&\quad \circ \ frag_{\pi_{date}} \circ decrypt_{\mathsf{heq},name} \circ crypt_{\mathsf{heq},name}
\end{aligned}$$

decrypt commutes with $frag$, must be applied to both fragments and it can be simplified when components are missing in the fragment

$$\begin{aligned}
&\equiv count \circ defrag_{\pi_{date}} \\
&\quad \circ \ (group_{date} \circ \pi_{date} \circ \sigma_{(today-date<8)}, \\
&\qquad \sigma_{(address=desk)} \circ decrypt_{\mathsf{heq},name}) \\
&\quad \circ \ frag_{\pi_{date}} \circ crypt_{\mathsf{heq},name}
\end{aligned}$$

decrypt commutes with σ, the predicate of σ does not work on encrypted data, thus it does not require homomorphic encryption to compute selection

$$\begin{aligned}
&\equiv count \circ defrag_{\pi_{date}} \\
&\quad \circ \ (group_{date} \circ \pi_{date} \circ \sigma_{(today-date<8)}, \\
&\qquad decrypt_{\mathsf{heq},name} \circ \sigma_{(address=desk)}) \\
&\quad \circ \ frag_{\pi_{date}} \circ crypt_{\mathsf{heq},name}
\end{aligned}$$

decrypt commutes with *defrag*

$$\begin{aligned}
&\equiv count \circ decrypt_{\mathsf{heq},name} \circ defrag_{\pi_{date}} \\
&\quad \circ \ (group_{date} \circ \pi_{date} \circ \sigma_{(today-date<8)}, \sigma_{(address=desk)}) \\
&\quad \circ \ frag_{\pi_{date}} \circ crypt_{\mathsf{heq},name}
\end{aligned}$$

Finally *count* computes the number of groups but it does not rely on the value of tuples, so they do not require to be decrypted

$$\begin{aligned}
&\equiv count \circ defrag_{\pi_{date}} \\
&\quad \circ \ (group_{date} \circ \pi_{date} \circ \sigma_{(today-date<8)}, \sigma_{(address=desk)}) \\
&\quad \circ \ frag_{\pi_{date}} \circ crypt_{\mathsf{heq},name} \ \blacksquare
\end{aligned}$$

3.3 Laws for Composition

We now generalize our transformational approach by formally-defined composition laws following Backus's approach [Bac78]. They specify how protection and query functions interact, in particular how they commute. We review them briefly.

First, the *Identity Laws* specify that pairs of protection functions are inverse to each other. Applying a protection then discarding a protection results in no protection. When oriented from left to right, these three rules can be used to introduce privacy functions in a query:

$$id \equiv decrypt_{\mathsf{s},as} \circ crypt_{\mathsf{s},as} \tag{1}$$
$$id \equiv defrag_{\pi_{as}} \circ frag_{\pi_{as}} \tag{2}$$
$$id \equiv defrag_{\sigma_p} \circ frag_{\sigma_p} \tag{3}$$

Second, all other rules specify when protection-discarding and query-functions commute. They can be used to delay the discarding of protection (*i.e.*, "push" $defrag/decrypt$ to the left in a query); intuitively, this means that more computations are performed in the cloud rather than at the owner's site.

The *Projection Laws*, (4) specifies that projection and *decrypt* commute. When fragmentation is used, laws (5)-(8) specify a projection becomes a pair of projections (one per fragment):

$$\pi_a \circ decrypt_{\mathsf{s},a} \equiv decrypt_{\mathsf{s},a} \circ \pi_a \tag{4}$$
$$\pi_{a\bar{a}} \circ defrag_{\pi_a} \equiv defrag_{\pi_a} \circ (\pi_a, \pi_{\bar{a}}) \tag{5}$$
$$\pi_a \circ defrag_{\pi_a} \equiv defrag_{\pi_a} \circ (\pi_a, \pi_a) \tag{6}$$
$$\pi_{\bar{a}} \circ defrag_{\pi_a} \equiv defrag_{\pi_a} \circ (\pi_{\bar{a}}, \pi_{\bar{a}}) \tag{7}$$
$$\pi_a \circ defrag_{\sigma p} \equiv defrag_{\sigma p} \circ (\pi_a, \pi_a) \tag{8}$$

Grouping Laws (9),(10) specify that *decrypt* and *group* commute. When groups are based on encrypted components, *group* must take into account encryption (10). Vertical fragmentation and *group* commute, when groups can be computed in a single fragment (11),(12). Horizontal fragmentation and *group* commute and groups must be computed in both fragments (13):

$$group_a \circ decrypt_{\mathsf{s},b} \equiv decrypt_{\mathsf{s},b} \circ group_a \quad \text{if } a \notin (b) \tag{9}$$
$$group_a \circ decrypt_{\mathsf{s},b} \equiv decrypt_{\mathsf{s},b} \circ group_{\mathsf{s}_a} \text{if } a \in (b) \tag{10}$$
$$group_a \circ defrag_{\pi_a} \equiv defrag_{\pi_a} \circ (group_a, id) \tag{11}$$
$$group_{\bar{a}} \circ defrag_{\pi_a} \equiv defrag_{\pi_a} \circ (id, group_{\bar{a}}) \tag{12}$$
$$group_a \circ defrag_{\sigma_p} \equiv defrag_{\sigma_p} \circ (group_a, group_a) \tag{13}$$

Selection Laws are quite similar to *Grouping Laws*, but the different cases are based on predicate parts dealing with one fragment, the other or both. In particular, (16) requires $\sigma_{pa\bar{a}}$ to select tuples in both vertical fragments *after* the defragmentation (*i.e.*, at owner's place):

$$\sigma_p \circ decrypt_{\mathsf{s},a} \equiv decrypt_{\mathsf{s},a} \circ \sigma_p \quad \text{if } dom(p) \notin \mathscr{P}(a) \tag{14}$$
$$\sigma_p \circ decrypt_{\mathsf{s},a} \equiv decrypt_{\mathsf{s},a} \circ \sigma_{\mathsf{s}_p} \text{ if } dom(p) \in \mathscr{P}(a) \tag{15}$$
$$\sigma_{pa\wedge p\bar{a}\wedge pa\bar{a}} \circ defrag_{\pi_a} \equiv \sigma_{pa\bar{a}} \circ defrag_{\pi_a} \circ (\sigma_{pa}, \sigma_{p\bar{a}}) \tag{16}$$
$$\sigma_{pa\wedge t\wedge t} \circ defrag_{\pi_a} \equiv defrag_{\pi_a} \circ (\sigma_{pa}, id) \tag{17}$$
$$\sigma_{t\wedge p\bar{a}\wedge t} \circ defrag_{\pi_a} \equiv defrag_{\pi_a} \circ (id, \sigma_{p\bar{a}}) \tag{18}$$
$$\sigma_{p'} \circ defrag_{\sigma p} \equiv defrag_{\sigma_p} \circ (\sigma_{p'}, \sigma_{p'}) \tag{19}$$

In law (20), *count* does not access values so that *decrypt* can be discarded:

$$count \circ decrypt_{\mathsf{s},as} \equiv count \tag{20}$$

Finally, *Protection Composition Laws* commute and distribute functions in order to apply previous laws:

$$f \circ id \equiv id \circ f \equiv f \tag{21}$$
$$(f1, f2) \circ (g1, g2) \equiv (f1 \circ f2, g1 \circ g2) \tag{22}$$
$$frag_{\pi_a} \circ decrypt_{\mathsf{s},a} \equiv (decrypt_{\mathsf{s},a}, id) \circ frag_{\pi_a} \tag{23}$$
$$frag_{\pi_a} \circ decrypt_{\mathsf{s},\bar{a}} \equiv (id, decrypt_{\mathsf{s},\bar{a}}) \circ frag_{\pi_a} \tag{24}$$
$$decrypt_{\mathsf{s},a} \circ defrag_{\pi_a} \equiv defrag_{\pi_a} \circ (decrypt_{\mathsf{s},a}, id) \tag{25}$$
$$decrypt_{\mathsf{s},\bar{a}} \circ defrag_{\pi_a} \equiv defrag_{\pi_a} \circ (id, decrypt_{\mathsf{s},\bar{a}}) \tag{26}$$
$$frag_{\sigma_p} \circ decrypt_{\mathsf{s},a} \equiv (decrypt_{\mathsf{s},a}, decrypt_{\mathsf{s},a}) \circ frag_{\sigma_p} \tag{27}$$
$$decrypt_{\mathsf{s},a} \circ defrag_{\sigma_p} \equiv defrag_{\sigma_p} \circ (decrypt_{\mathsf{s},a}, decrypt_{\mathsf{s},a}) \tag{28}$$

4 Implementation

The functional DSL for the composition of privacy-aware query-based applications introduced above permits the definition of Functional Programming (FP)-like equivalence laws and the transformation of a privacy-preserving local application into a privacy-preserving cloud application. In this section, we present our Scala-based [OSV08, CB14] framework[1], a prototype that makes it possible to program sophisticated privacy-aware cloud applications. In particular, our implementation satisfies the laws introduced previously and harnesses Scala's type system to make application conditions explicit that are required by the laws.

4.1 From Theory to Practice

One of the main differences between the DSL and the actual implementation is that we distinguish functions that compute queries from functions that change the shape of the database. In the previous section, the language uses the pointwise application of one function to the result of another, which successively reduces the database until the result is obtained. This abstraction helps reasoning on the query design. However, it misses practicality for real world programming since we do not want to reduce our database and lose components. In practice, two different levels of computation are more useful: one that uses database components to compute queries, and another one that modifies the database shape and applies protection.

The programs in Fig. 3 query the number of meetings per day at the desk last week, for a local (a) and a cloud (b) application. The local application only uses functions to query the database (σ, π, `group` and `count`). In contrast, the cloud application also composes functions for protection (`crypt` and `fragV`).

The Guardian Monad. In functional programming, functions with side effects such as protection functions are performed in a *monad* [Wad92a], a pattern that makes it easy to chain function calls. In particular, FP deals with side effects using a *state monad* [Wad92b] that attaches state information to function calls. For this reason, our framework provides a state monad suitable for our purposes. Our state monad, called *guardian*, is defined based on the following requirements:

- The state is the database.

[1] The sources of our framework and examples are available on the Github platform at `https://github.com/rcherrueau/phant`

```
1  for {
2    _ <- configure[Date,Name,Addr]
3    q <- query (db => {
4           val r1 = σ (db) (lastweek
5                           ∧ atdesk)
6           val r2 = π (r1) (date)
7           val r3 = group (r2) (date)
8           val r4 = count (r3); r4
9         })
10 } yield q
```

(a) Local Application

```
1  for {
2    _  <- configure[Date,Name,Addr]
3    _  <- crypt (_2) (HEq(_))
4    _  <- fragV (_1) (Site1(_), Site2(_))
5    qL <- queryL (fragL => {
6            val r1 = σ (fragL) (σ_lift lastweek)
7            val r2 = π (r1) (π_lift date)
8            val r3 = group (r2) (date); r3
9          })
10   qR <- queryR (fragR => {
11           val r1 = σ (fragR) (σ_lift atdesk)
12           val r2 = π (r1) (id); r2
13         })
14 } yield count (gather (qL, qR))
```

(b) Cloud Application

Figure 3: Privacy-Preserving Agenda as Local (a) and Cloud (b) Application

- Query functions only access the components of the database. The application of a query function returns some result that can be used as input for a second query, without modifying the database. For instance, the selection in figure 3a accesses the content of the database and returns the result in `r1` (line 4). Values in `r1` are then used for the projection (l.5) and so on, until the `count` operation (l.7). At the end of the program (l.9), the `q` variable gets the result of the query. But, the database remains the same as in input.

- Protection functions only modify the shape of a database. For instance, the `crypt` instruction in figure 3b modifies the database by encrypting the second column with a homomorphic scheme that supports equality testing (l.3). Similarly, the `fragV` instruction splits the database vertically on the first column (l.4). It distributes the left fragment on site number one and the right fragment on site number two. Henceforth, the querying should be done on both fragments (l.5-13).

The protection functions `crypt` and `fragV` are complemented by discarding functions `decrypt` and `defragV`. The discarding functions from the framework differ from the ones in the DSL because the framework ones discard protection on the database, whereas the functions of the DSL discard protections on the query result. In the DSL example of the cloud application (Sec. 3.2), the call of $defrag_{\pi_{date}}$ joins at the client side the result of both fragments and hands it over to the *count* computation. In the corresponding implementation (Fig. 3b), calling a `defragV` will join fragments but not the result of the queries. For this reason, the framework provides the `gather` instruction, which is applied at the query level. It brings back data at the client side and discards protection. Given this, the program 3b gathers the result from both fragments in order to count the number of meetings (l.14). Note that in this example like in the example of the DSL, the `gather` does not have to decrypt names because the right query discards them.

Monadic programming and types. In example 3b it does not make sense to encrypt the second column twice. As it does not make sense to query the fragmented database in the same manner as the local database. Furthermore, it does not make sense to group on an encrypted data that does not support equality testing. To put it simply, if privacy techniques are to be composed, errors can easily be introduced that yield to runtime errors!

A main feature of monads is the use of types to exhibit what it means to execute chained function applications. Given this, the guardian monad exhibits useful information at the type level to help the programmer write programs that cannot go wrong. For instance, the type of the guardian at the end of program 3b is:

```
Guard[
  Site0[DB[Raw[Date] |: Raw[Name] |: Raw[Addr]]],                          // (1)
  (Site1[DB[Raw[Date] |: Id]], Site2[DB[HEq[Name] |: Raw[Addr] |: Id]]), // (2)
  Site0[List[Int]]]                                                       // (3)
```

With:

(1) The shape of the database at the start of the computation. Here the guardian only accepts, as input, database that stores date, name and address (*i.e.*, `DB[Date |: Name |: Addr]`) in plain form (*i.e.*, `Raw`) and uploaded at client side (*i.e.*, `Site0`).

(2) The shape of the database at the end of the computation. Here the guardian transforms the database into two fragments. The first one stores dates in plain form at site one. The second one stores names and addresses at site two. The guardian also encrypts names with a homomorphic encryption that supports equality (*i.e.*, `HEq`).

(3) The type of the query result. Here, the type of the number of meetings per day at the desk last week (*i.e.*, `List[Int]`). The `Site0` annotation informs the developer that a part of the query is computed at the client side.

```
for {
  _ <- configure[Date,Name,Addr]
  _ <- crypt (_2) (HEq(_))
  // ill-typed, encryption of
  // all-ready encrypted column:
  _ <- crypt (_2) (HEq(_))
  // ...
} yield ()
```

Figure 4: Twice encryption does not type check

```
for {
  _ <- configure[Date,Name,Addr]
  _ <- crypt (_2) (HEq(_))
  _ <- fragV (_1) (Site1(_), Site2(_))
  // ill-typed, query on a non-local
  // database.
  q <- query (db => { /* ... */ })
  // ...
} yield ()
```

Figure 5: Fragmentation requires querying on fragments

A guardian uses type information to ensure that compositions cannot go wrong and gives useful information at compile time. Hence, trying to encrypt an already encrypted column does not type check (Fig. 4). Querying a fragment with a local approach does not type check (Fig. 5). Grouping/Filtering on an encrypted data that does not support the test does not type check (Fig. 6). Generally speaking, the implementation satisfies the laws of section 3.3. We rely on property-based testing with ScalaCheck [Nil07] to argue their correctness.

Finally, monad bindings enable the naming of query result such as in program 3b. Value `qL` contains the result of the left query. Value `qR` contains the result of the right query. Naming is essential when the programmer wants to implement a profitable strategy like the *left-first* strategy from figure 2. Program 7 implements the left-first strategy. It simply consists of naming the result of left fragment, and then use it in the right fragment.

4.2 Feedback on the Implementation with Scala

The Scala programming language is good for generalization. We harness an advanced use of Scala implicits and type members to perform type-level computations [dSOMO10]. This enables the definition of arity-polymorphic databases, so that the guardian monad can be implemented once

```
for {
  _ <- configure[Date,Name,Addr]
  // Symmetric encryption of Name.
  // Symmetric doesn't support equatlity
  // testing.
  _ <- crypt (_2) (Symmetric(_))
  q <- query (db => {
                // ill-typed, name doesn't
                // support equality testing
                groupby (db) (name)
              })
  // ...
} yield ()
```

Figure 6: Grouping on encrypted data requires support equality

and for all. Without an arity-polymorphic database we would have to write as many guardian monads as a database could contain attributes, just like for tuples. In the previous examples (*e.g.*, Fig. 3b), the presence of integers prefixed by an underscore is the direct consequences of type-level computation. Integers with an underscore are Church encodings of the natural numbers at the type level. They make it possible to identify, at compile time, which column has to be encrypted, and the type of both fragments after fragmentation.

Scala unifies functional and object-oriented programming. Under the hood, data are objects and operations method calls. This object model is good for modularity and generalization, for instance, with subtyping. However, it makes type inference less powerful than that of ML-like functional languages that use the Hindley-Milner algorithm. Because of that, the guardian monad sometimes requires to explicitly specifying the type to help the compiler infer type parameters. The code examples we presented above are a beautified version, omitting a few type annotations that are needed by the compiler, so that readers can more easily understand the intention of the guardian monad. The code with all necessary type annotation is available on the Github platform.

Finally, the guardian monad is a prototype, whose implementation proves its validity. In the future we intend to bind the current implementation with nice Scala libraries such as Akka[2] for the distribution and Slick[2]for the mapping with real relational databases.

5 Related Work

In the following, we compare our work to three sets of related work: approaches that focus on data fragmentation (but also including encryption), related work providing language support for privacy properties and approaches, such as sticky policies and security-aware objects that enable privacy properties to be expressed and enforced directly.

Data fragmentation is a recent technique that strives to ensure strong confidentiality properties without the query overhead of encryption-based approaches. A recent overview [dVEF+13] presents a wide range of fragmentation techniques and corresponding algorithms. However, none of the discussed approaches includes, as our approach provides, declarative means for the construction of fragmentation algorithms, their composition especially with encapsulation techniques and formal property guarantees over implementations.

Several language-based approaches have been proposed for other privacy properties. Tetali *et*

[2] `http://akka.io/`; `http://slick.typesafe.com/`

```
for {
  _   <- configure[Date,Name,Addr]
  _   <- crypt (_2) (HEq(_))
  _   <- fragV (_1) (Site1(_), Site2(_))
  // Queries on left fragment to get identifiers
  // of meetings the past week:
  ids <- queryL (fragL => {
                   val r1 =
                     σ (fragL) (σ_lift lastweek)
                   val r2 = π (r1) (id); r2
                 })
  q   <- queryR (fragR => {
                   // Reduces the number of
                   // elems with ids of left
                   val r1 = σ (fragR) {
                     case (_,_,id) =>
                       ids.exists(id)
                   }
                   val r2 = group (r1) (name)
                   val r3 = count (r2); r3
                 })
} yield q
```

Figure 7: Number of people Alice met last week – left-first strategy

al. [TLMM13], *e.g.*, have proposed analysis techniques for compositions of different types of homomorphic encryption algorithms. Fournet *et al.* [FKDL13] define ZQL, a query language operating over annotated database schemas that use strong typing in order to ensure security properties of generated implementations in F# and C++. Reed and Pierce [RP10] propose a specialized type system to enforce privacy guarantees by means of differential privacy. However, none of these approaches provides language support for the composition of fragmentation and encapsulation techniques.

Another domain of related work consists in support for the expression of privacy properties in the form of policies and constraints over accesses to runtime objects. Sticky policies [KSW02], *e.g.*, represent a class of policies that enable the abstract definition of privacy properties and their enforcement through runtime annotations. Self-protecting software systems [YEM14], such as self-defending objects [HCR04], support the protection of privacy properties of runtime entities by strong encapsulation and access control of these entities. Again however, none of these approaches, as well as other policy-based and encapsulation-based privacy techniques, support properties involving the composition of fragmentation and encapsulation techniques.

6 Conclusion

In this paper we have addressed the problem of how to define and enforce privacy properties in the context of complex computations executed in mutualized environments, such as the Cloud, and that require different privacy-enforcing techniques to be used. We have considered privacy properties that are formulated in terms of compositions of data fragmentation and encryption.

We have provided programming language support for the declarative definition of such composed privacy-enforcement strategies and an implementation on top of the Scala language that, using a specialized type system, ensures privacy properties by construction. We have also provided a set of laws that ensure privacy properties and that are satisfied by the language mechanisms we provide.

As future work we intend to extend the set of fragmentation techniques and approaches to encryption in order to obtain a full-fledged composition theory for these two classes of privacy-enforcing techniques.

References

[ABG+05] Gagan Aggarwal, Mayank Bawa, Prasanna Ganesan, Hector Garcia-Molina, Krishnaram Kenthapadi, Rajeev Motwani, Utkarsh Srivastava, Dilys Thomas, and Ying Xu. Two can keep A secret: A distributed architecture for secure database services. In *CIDR*, pages 186–199, 2005.

[Bac78] John W. Backus. Can programming be liberated from the von neumann style? A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[CB14] Paul Chiusano and Rnar Bjarnason. *Functional Programming in Scala*. Manning Publications Co., 2014.

[CJL12] Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. A software-hardware architecture for self-protecting data. In *ACM Conference on Computer and Communications Security*, pages 14–27, 2012.

[dSOMO10] Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 341–360, 2010.

[dVEF+13] Sabrina De Capitani di Vimercati, Robert F. Erbacher, Sara Foresti, Sushil Jajodia, Giovanni Livraga, and Pierangela Samarati. Encryption and fragmentation for data confidentiality in the cloud. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, pages 212–243, 2013.

[FKDL13] Cédric Fournet, Markulf Kohlweiss, George Danezis, and Zhengqin Luo. Zql: A compiler for privacy-preserving data processing. In *USENIX Security*, pages 163–178, 2013.

[HCR04] John W. Holford, William J. Caelli, and Anthony W. Rhodes. Using self-defending objects to develop security aware applications in java. In *ACSC*, pages 341–349, 2004.

[KSW02] Günter Karjoth, Matthias Schunter, and Michael Waidner. Platform for enterprise privacy practices: Privacy-enabled management of customer data. In *Privacy Enhancing Technologies*, pages 69–84, 2002.

[Nil07] Rickard Nilsson. ScalaCheck: Property-based testing for Scala, 2007.

[NLV11] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, pages 113–124, New York, NY, USA, 2011. ACM.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.

[PM11] Siani Pearson and Marco Casassa Mont. Sticky policies: An approach for managing privacy across multiple parties. *IEEE Computer*, 44(9):60–68, 2011.

[RP10] Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *ICFP*, pages 157–168, 2010.

[Sal03] David Salomon. *Data privacy and security: encryption and information hiding*. Springer Science & Business Media, 2003.

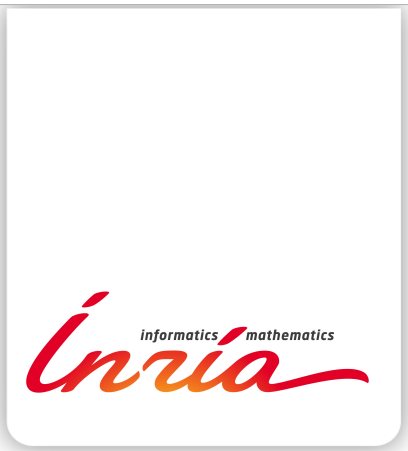
[TLMM13] Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd Millstein. MrCrypt: static analysis for secure cloud computations. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications OOPSLA*, pages 271–286. ACM, 2013.

[WABL+08] Daniel J. Weitzner, Harold Abelson, Tim Berners-Lee, Joan Feigenbaum, James A. Hendler, and Gerald J. Sussman. Information accountability. *Commun. ACM*, 51(6):82–87, 2008.

[Wad92a] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.

[Wad92b] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pages 1–14, 1992.

[YEM14] Eric Yuan, Naeem Esfahani, and Sam Malek. A systematic survey of self-protecting software systems. *TAAS*, 8(4):17, 2014.



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399